Escape Roomba: Evaluating Solution Methods for POMDPs

Allison Lettiere, Ana Sofia Nicholls, and Ali Saaed
Stanford University, Stanford, CA, 94305

This paper examines the effect of partially observable Markov decision process (POMDP) algorithm selection on effectively navigating a Roomba robot vacuum to a target location. We examined two Markov decision process (MDP) methods - Discrete Value Iteration and Monte Carlo Tree Search (MCTS)- and two POMDP evaluation methods - Randomized Point-Based Value Iteration (RPBVI) and Partially Observable Monte Carlo Planning With Observation Widening (POMCPOW). The POMDP implementations established for the Roomba have applications reaching beyond the specific constraints of this problem. The Roomba's accumulation of knowledge applies to other autonomous and semi-autonomous systems, extending to issues ranging in importance and financial scale from Pursuit-Evasion games to spacecraft decision making. By continuing to explore the effectiveness of various MDP and POMDP methods, we are equipped with a toolbox to make more informed policy-building decisions in the future. Our research provides empirical, quantitative results that signify the effectiveness of each of the methods. Of the three algorithms, Discrete Value Iteration was the most efficient when examining overall reward and execution time. This indicates that the support provided in the underlying MDP was sufficient to converge to a successful solution for the Roomba problem.

I. Key Concepts

A. MDP

A MDP is a system in which an agent navigates through different states of the system by choosing actions and collecting rewards based on state-action pairs. MDPs rely on the Markov assumption that the next state relies solely upon the current state and action, and not on any of the previous actions or states.

B. POMDP

A POMDP is an MDP in which there is state uncertainty. To address this uncertainty, POMDPs make use of belief states, which represent the probability that an agent is in each state. The models in POMDPs thus rely on belief states instead of actual states. In a POMDP there is also an additional model: the observation model, $O(o \mid s, a)$ which represents the probability of observing observation o given state s and action a.

C. Alpha Vectors

An alpha vector represents the utility of an action from each possible state. There is one alpha vector per action. The utility of an action is provided by the dot product of the alpha vector with the vector of the current belief state. Thus, optimal utility is given by $U^*(b) = \max_\alpha \alpha^\intercal b$.

D. Offline vs. Online Methods

Offline methods entail performing a majority of the computation before run time. Offline methods may necessitate significantly more storage and computation time to produce a policy, and thus are not preferred for large state-action spaces. Online methods compute policies based on the current belief state, and thus only consider reachable belief states. While online methods require significantly more computation per step while running, these methods are well suited for problems with high-dimensionality or prohibitively large belief spaces.

II. Introduction

Our study addresses the navigation of a Roomba robot vacuum to a target location (outside the room). The Roomba's environment contains walls and a staircase. If the Roomba finds itself at the staircase, it will fall and suffer fatal damage.

Collisions with the wall also cause damage, although the Roomba can survive these collisions. The Roomba knows the layout of the environment, however it does not know where inside the environment it is located. The robot has a bumper sensor, which indicates whether or not the Roomba has had contact with a wall.

The goal at the beginning of the study was to attempt different solution methods for MDPs and POMDPs and evaluate which method maximized utility. The study included four approaches to navigating the Roomba to its target location: Discrete Value Iteration, Monte Carlo Tree Search, Randomized Point-Based Value Iteration, and POMCPOW.

To answer these questions, our study collected data on the rewards amassed with each approach. In addition, we also documented the runtime for each implementation. This study provides an adaptation of POMDP theory that can illuminate the nuances of various MDP and POMDP solution methods. By working directly with these methods in the Roomba environment, we can apply our learning to problems with similar dimensionality and uncertainty.

III. Relevant Literature

The Roomba simulation is an example of an autonomous system: "it can construct and execute a plan to achieve its assigned goals, without human intervention" [1]. However, in real life this type of robotic vacuum should be considered a semi-autonomous system. If the Roomba requires relocation to another floor, it would necessitate human assistance. An interesting extension to our problem would involve producing a policy that addresses the collaboration between the device and the human user and outlines the resulting uncertainty.

Our problem examines a context much like that outlined in the research paper *Building Strong Semi-Autonomous Systems* by Shlomo Zilberstein [2]. Their work outlines how uncertainty, in a robot situation, may be discrete. The authors present the notion that an agent must take into account its own degree of uncertainty. This scenario, similar to that of our Roomba, is a "a planning problem: given a complete and correct model of the world dynamics and a reward structure, find an optimal way to behave[2]."

When examining the models implemented in our experiment, we can look at additional work that has been done to formulate and extend some of these algorithms.

The Value Iteration algorithm has often presented two dilemmas: history and dimensionality. While these two issues may limit the expansion of the algorithm to real world problems, the academic work, *Point-based value iteration: An anytime algorithm for POMDPs*, [3] indicates that if the curse of history can be avoided, many real world POMDPs may not suffer the curse of dimensionality. This is applicable to higher level POMDPs and larger-scale implementations of the Roomba.

An extended approach to Point-Based Value Iteration, Perseus, is outlined in [4]. Perseus uses approximate value backup stages and only backs up a randomly selected subset of points in the belief set. With this algorithm, a single backup may improve the value of many points in the set.

The Department of Computer Science at Ben-Gurion University has developed an implementation of Forward Search Value Iteration[5] (one of the algorithms mnot implemented during the course of our research). The algorithm traverses both the belief space and the underlying MDP. Forward Search Value Iteration outperformed both Heuristic Search Value Iteration and Point-Based Value Iteration. A major takeaway from their academic inquiry is that "given that different approximation algorithms for POMDPs are likely to perform well on different classes of domains...[one should] determine prior to execution of an algorithm whether it will work well on a given problem[5]." This is a key takeaway that should travel into any future work regarding POMDPs and scenarios with uncertainty.

Lastly, a group of Brazilian computer scientists conducted research that uses a POMDP model to approach robotic uncertainty with a solution that combines both the notion of localization with the overall mission of the vacuum. Their study, Cleaning Task Planning for an Autonomous Robot [6], outlines these objectives and results.

IV. Approach

Our study aims to contrast the behaviors and performances of four disparate algorithmic methods for the development of policies to navigate a Roomba out of a room. Our algorithms are subdivided into two MDP methods (one offline, one online) and two POMDP methods (one offline, one online). Respectively, these algorithms are 1) Discrete Value Iteration, 2) MCTS, 3) RPBVI, and 4) POMCPOW.

These four methods were tested over a simulated Roomba/room environment implemented in Julia. The room features a polygonal perimeter, one segment of which is designated the goal state, another segment of which is designated the stairwell (failure). The Roomba is defined by its state, which contains four fields: its x location (meters), y location (meters), orientation (radians), and a boolean indicator as to whether the robot has reached the goal state or stairwell. At

any non-terminal state, the Roomba must take an action. This action is a movement, defined by its linear velocity (m/s) and angular velocity (rad/s) over a time step (s). A policy for the Roomba is therefore a mapping from each state to an action. The Roomba observes its environment via a bump sensor which indicates whether the device is in contact with the perimeter. At the beginning of a simulation, the Roomba is randomly placed in a non-terminal state.

This Roomba environment is encoded using POMDPs.jl, a package which provides a core interface for working with MDPs and POMDPs. Our code interacts with underlying Roomba MDP and POMDP objects built around the aforementioned specifications. Three of our four methods adapt solvers provided alongside the POMDPs.jl package, namely Discrete Value Iteration, MCTS, and POMCPOW.

All four methods implement the same reward function that provides a time based penalty (-0.1 / time step), a penalty when the Roomba collides with a wall (-1) and a larger, terminating penalty if the Roomba falls down the stairs (-10). A large positive reward is provided for reaching the goal state (+10). In our experimental procedure, one trial consists of five simulations of a Roomba journey starting on different random seed, navigating towards the goal using the policy we are testing. The average reward accrued over the five simulations yields the mean total reward, which is the metric by which we quantize the efficacy of our methods. Additionally, we keep track of the amount of time required for our models to train (if offline) or run (if online), so we can compare the efficiency of the methods.

V. Evaluated Methods

A. Baseline Policy

As our first step, we implemented a semi-random policy formation. This involved initially spinning for ten time steps to randomize the orientation while maintaining constant velocity. If the Roomba bumped into the wall, we generated a random angle and produced another action. The Roomba performed random actions until it reached the exit of the environment. As a means of evaluation, we simulated this policy five times and found the mean reward.

While this initial policy was evidently not optimal (with an average reward of -19.46), the Roomba did reach the goal, and thus it serves as a baseline to compare the effectiveness of the other policy building strategies implemented in our project.

B. Modeling the POMDP as an MDP

We experimented with two MDP methods on this POMDP problem. First, we discretized both the action and the state spaces using the functions provided in the starter files, some of which we modified, so that we could initialize the MDP that would model the Roomba problem. Next, two approaches were tested using this MDP: Discrete Value Iteration and MCTS.

1. Discrete Value Iteration

Discrete Value Iteration is an exact solution method for MDPs. In Value Iteration, we compute the optimal value function $U*$ by first initalizing Uo to be zero for all states in the MDP, then iterating and computing U_{k+1} for all states using the Bellman Equation: $U_{k+1}(s) = \max_a (R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_k(s'))$. Once $U*$ converges or we reach our maximum number of iterations, we extract the policy from $U*$ using $\pi(s) \leftarrow \arg\max_a (R(s,a) + \gamma T(s' \mid s,a) U^*(s'))$.

The solution using Discrete Value Iteration was implemented using DiscreteValueIteration.jl. The two parameters for the solver were the maximum number of iterations and the Bellman residual. We tested different numbers of iterations, keeping the Bellman residual constant at 0.000006. A surprising result was that this Value Iteration solver converged on the optimal policy rather quickly: after 3 iterations, the utility plateaued.

Number Of Iterations	Reward(5 trials, average)	Total Solving Time (seconds)
1	-10.02, -10.03, 8.54, 8.55, 8.5, **1.100**	52.908870 seconds
2	7.92, 7.13, 8.54, 8.55, 8.5, **8.100**	30.380268 seconds
3	7.92, 8.13, 8.54, 8.55, 8.5, **8.300**	74.872632 seconds
10	7.92, 8.13, 8.54, 8.55, 8.5, **8.300**	179.141849 seconds
100	7.92, 8.13, 8.54, 8.55, 8.5, **8.300**	121.338804 seconds

Fig. 1 Discrete Value Iteration Run with Varying Number of Iterations

2. Monte Carlo Tree Search

The next approach we implemented was Monte Carlo Tree Search (MCTS) using MCTS.jl. MCTS is an online approach that runs simulations from the current state to update the state-action value function Q(s, a). For this method, there are three stages: search, expansion stage, and rollout.

We define the data structure, T, to be an initially empty set which will be filled with states we are exploring. The first stage is the search stage. If the current state s is an element of T, we proceed with the search stage (if s is not in T, then we move on to the expansion stage). We update $Q(s, a)$ for all the state-action pairs we have tried. While searching, we keep track of a count $N(s, a)$ for each state-action pair we have tried. While searching, we execute the action which maximizes $Q(s, a) + c\sqrt{\log N(s)/N(s, a)}$. $N(s)$ can be obtained by summing over $N(s, a)$. The parameter c is chosen according to the amount of exploration desired. The second term altogether is called an exploration bonus, which promotes actions that have been visited less often.

The next stage is the expansion stage, which is reached when the current state is not an element of T. From this state, we iterate over all of the current state's available actions and define $N(s, a)$ and $Q(s, a)$, which can be initialized to zero or some other value if one wishes to incorporate prior knowledge about the problem. The current state is then added to T.

The last stage is the rollout stage. We choose actions with a default policy (rollout) until we reach the desired depth. This policy does not have to be optimal but can incorporate prior knowledge if we know that one region of the state space has a higher utility than another. We return the expected value and this is used to update $Q(s, a)$ in the search.

Usually we complete the algorithm when we have reached the desired number of iterations, but other requirements for halting the algorithm can also be incorporated. To determine the best action at a given step, we choose an action that maximizes $Q(s, a)$ then rerun MCTS again to choose the next action. We can recycle the $N(s, a)$ and $Q(s, a)$ when we rerun MCTS.

The chart below contains data for the rewards accrued with different parameters (number of iterations, exploration constant, and depth) and the runtime of each parametrization.

Number Of Iterations	Exploration Constant	Depth	Rewards (5 trials, average)	Total Solve Time (seconds)
50	5	20	-10.2, -10.3, 8.54, 8.45, **1.3**	0.17 seconds
50	1	20	-10.02, -10.03, 8.44, 8.25, 8.5, **1.028**	0.25 seconds
50	5	30	-10.02, -10.03, 8.54, 8.15, 8.4, **1.000**	0.000013 seconds
100	5	20	-10.02, -10.03, 8.44, 8.35, 8.5,**1.040**	0.000014

Fig. 2 Monte Carlo Tree Search Run with Varying Parameters

C. POMDP Methods

1. Randomized Point-Based Value Iteration

The first POMDP method implemented was Randomized Point-Based Value Iteration, based upon Section 6.4.4 of *Decision Making Under Uncertainty* [7]. This method was implemented using some of functions provided in the source code, modified to fit our needs. Our POMDP method was built from scratch save for five helper functions from the starter files. We began by initializing the first alpha vector as a lower bound. Each component of this vector is set to the value of the reward produced by taking the best action from the state in the environment that yields the smallest potential rewards. For the Roomba, this is the reward provided by going from the corner directly next to the stairs to the end goal (room exit zone - provided by the function *get_goal_xy*) This computation was based off of the formula $\max_a \sum_{t=0}^{\infty} \gamma^t \min_s R(s, a) = \frac{1}{1-\gamma} \max_a \min_s R(s, a)$ with the reward established above and a discount factor $\gamma = 0.90$.

A collection of 100 possible beliefs were extracted from the total number of beliefs by iterating through the belief updater and resampling points within the belief space using the particle filter updater. After the sampled belief vector was built, *RandomizedPointBasedUpdate* was called on the belief vector and initial alpha vector. This function returns a collection of alpha vectors, each associated with an action, to use during the simulation. The alpha vectors are the basis of the policy. The algorithm within randomized point-based update randomly extracts beliefs from the pre-compiled set of 100 and determines an alpha vector for this belief using a backup function. The backup function loops through each of the discretized actions and forms a dictionary linking each (action, observation) pair to an optimized alpha vector. The function then extracts the next state given the collision status of the current mean of the belief state and action. A dictionary mapping state-action pairs to alpha vectors is updated for each state in the state space using the following equation and a discount factor of 0.9: $\alpha_a(s) \leftarrow R(s, a) + \gamma \sum_{s',o} O(o \mid s', a) T(s' \mid s, a) \alpha_{a,o}(s')$.

The alpha vectors related to specific action-observation pairs are represented using a dictionary that maps action, observation keys to alpha vector components.

Our implementation varies from the outline in *Decision Making Under Uncertainty* [7] to ensure that all of the discrete actions are accounted for once the offline computation is completed. Each belief backup produces a series of alpha vectors (one for each action). During the randomized point-based update, we check if the new alpha vector for a given action is included in the prior collection of alpha vectors. If it is not in the set, it is added to the collection. If the new collection already contains the action, we check whether the new alpha vector associated with this action improves the utility for the belief as compared to the utility for the belief given by the old alpha vector. If it does, the alpha vector for that action is updated. If it does not, the alpha vector that dominates at that belief for that action is maintained in the new collection.

During simulation, the Roomba chooses an action for a particular belief state by looping through the possible alpha vectors produced during the offline procedure. The action is selected according to the alpha vector that fulfills the following equation: $U^{\Gamma}(\mathbf{b}) = \max_{\alpha \in \Gamma} \alpha^{\top}\mathbf{b} = \max_{\alpha \in \Gamma} \sum_s \alpha(s)b(s)$.

The algorithm was tested varying number of updates to the function call: "gamma = RandomizedPointBasedUpdate(B, gamma)" (iterations). The results below indicate the the reward of running the simulation based upon the alpha vectors produced by a varied number of iterations. Only three trials were executed given the large computational run time.

Number Of Iterations	Reward (average)	Total Alpha Vector Generation Time (seconds)
1	6.16, 6.52, 6.56 **6.41**	80.154
2	6.25, 6.78, 6.65 **6.56**	161.754
3	6.45, 6.54, 6.39 **6.46**	241.388
5	6.78, 6.55, 6.42 **6.58**	409.618
10	7.62, 8.04, 7.58 **7.75**	869.301
25	8.60, 8.70, 8.60 **8.63**	2382.244

Fig. 3 Point Based Value Iteration Execution Rewards and Time

2. POMCPOW

Before introducing POMCPOW, we will provide background on Partially Observable Monte Carlo Planning (POMCP). POMCP is an online method that uses state trajectory simulations to construct POMDP trees representing a history proceeding from the current belief to actions/observations. At each action node, the rewards accrued from the simulations are used to estimate the Q value. When action and/or state spaces are continuous, such as in our case, we can modify POMCP with a technique called double progressive widening in which the number of children of a node in our tree is limited to kN^{α}, where N is the number of times the node has been visited and k and α are hyperparameters.

However, POMCP with double progressive widening (POMCP-DPW) is suboptimal for problems with a high value of information and exploration, particularly with continuous observations. While our Roomba environment is sufficient to run POMCP, we wanted to explore the better-suited POMCPOW method, which addresses the shortcomings of POMCP-DPW. Details about this algorithm can be found in the paper in which its introduced, authored by Zachary Sunberg et al.[8] In this method, weighted belief updates expand as more simulations are added, and beliefs more likely to be reached by an optimal policy have more particles.

We implemented a naive version of this policy using its solver from the POMDPs.jl package, hoping to optimize the hyperparameters for our environment. Preliminary results appeared promising, with the Roomba able to navigate out of the room when we discretized the state and action spaces. However the solver performed slowly, limiting our test to the default configuration using our experimental procedure. We found mean total rewards of -9.500 after running our algorithm for over 2.5 hours. While these results were suboptimal, the method outperformed the control policy. It should be noted that these results should not be considered a conclusive evaluation of POMCPOW.

VI. Discussion

We will now analyze the efficacy and efficiency of our four methods: 1) Discrete Value Iteration, 2) MCTS, 3) RPBVI, and 4) POMCPOW.

We implemented a semi-random policy as a control method for our experiment. The mean average reward accrued for this policy was **-19.46** units, which can be used as a baseline to reference our experimental algorithms.

1) Discrete Value Iteration is an offline MDP method providing an exact solution to the optimal value function from which we extract our policy. Hence, if we allow our solution to converge over an appropriate number of iterations, we are guaranteed to produce an optimal policy for the MDP. Note that the MDP was adapted from a discretized version of the POMDP. The solution reached may not be optimal for the POMDP. The reward for this method was precisely **8.30 units**. The value function converges quite efficiently over only three iterations, taking **74.87 seconds** of computation.

2) MCTS is an online MDP method seeking to update the state-action value function from the current state by exploring a policy tree. We can vary the number of iterations, the exploration constant, and the depth with which we search. We found that variation of the number of iterations between 50 and 100, variation of the exploration constant between 1 and 5, and variation of the depth between 20 and 30 all had minimal effect on the mean total rewards accrued from the policy, which averaged **between 1.00 and 1.30** units for our trials. There appeared to be no correlation between training time and these hyperparameter values, with all times **< 0.25 seconds**.

3) RPBVI is an offline POMDP method which seeks to update and optimize alpha vectors via random sampling. RPBVI allows varying the number of times the random point-based update is performed. We found a direct correlation between the number of iterations and reward accrued by the policy. Twenty-five iterations produced the with the highest mean total reward of **8.63 units**, after a training time of **2382 seconds**.

4) POMCPOW is an online POMDP method that improves upon POMCP's execution of the tree search. Our results for this method were inconclusive; we found that the POMDPs.jl implementation of POMCPOW ran too slowly, despite a discretized state and action space, to allow empirical optimization of the hyperparameters of the algorithm. In the test we were able to run, the policy generated accrued a mean total reward of **-9.50 units** with a run time of **> 2.5 hours**.

For the three policies with conclusive results, we found that RPBVI performed best; however, the computation took a significant amount of time. On the other hand, Discrete Value Iteration provided a policy that was nearly as efficacious with a much faster training time. Our study had the greatest trouble with the online methods of MCTS and POMCPOW. Despite running quickly, MCTS was unable to generate policies that consistently guided the Roomba to the goal, resulting in lower rewards, while POMCPOW ran prohibitively slowly, leading to inconclusive results.

Our Roomba simulation has relatively small dimensionality and complexity, so it is not surprising that the offline methods were most effective and efficient. For problems with similarly low complexity, we recommend offline methods. However, as the dimensionality and complexity increase, these offline methods may become infeasible. Our data suggests that the runtime of our most accurate method, RPBVI, is exponential in the number of iterations, whereas the runtime of MCTS, an online method, showed no correlation with reasonable variance of its hyperparameters.

VII. Conclusion

We aimed to learn about different solution methods and determine the best solution for the Roomba POMPD. We studied online and offline algorithms and experimented with MDP methods. The algorithms implemented were Discrete Value Iteration, MCTS, RPBVI, and POMCPOW. During our study, we also attempted to implement QMDP, Forward Search, AR-DESPOT (Anytime Regularized Determinized Sparse Partially Observable Tree), and Fast-Informed Bound which did not suit the Roomba POMDP. We have not discussed these to focus on comparing more optimal solutions.

Discrete Value Iteration had the best performance with an average reward of 8.30. This was surprising because modeling POMDPs as MDPs may cause a loss of information resulting from the difference in structure. Value Iteration's reward was lower than RPBVI, but converged significantly faster. RPBVI was the next most desirable solution, providing the highest mean reward of 8.63 despite a run time of about forty minutes. MCTS was less successful, converging on the solution very quickly but with an average reward of at most 1.30. Lastly, POMCPOW performed the worst with a runtime of over 2.5 hours and an average reward of -9.50. Though not all of these approaches performed optimally, they all performed much better than the random baseline policy which accrued an average reward -19.46.

This study directly compares solutions to an application of POMDP theory and revealed important information about the efficiency and optimality of different algorithms. Our study was not exhaustive and there may be better solutions to the Roomba POMDP and similar POMDPs with algorithms such as Lookahead with an Approximate Value Function, Branch and Bound, or Monte Carlo Value Iteration. The results obtained from studying the Roomba problem provide a basis for extrapolation to POMDPs with similar dimensionality and uncertainty, extending to other robotic systems or vehicles in which there are both highly rewarded and fatal actions. The Roomba is also emblematic of problems in which sensors provide intuition about the location of an agent. When applying these methods to problems with higher risks, it is important to understand that accuracy is an essential factor. Our models may not provide the accuracy required by larger-scale problems given that we extrapolated an MDP from the POMDP. Careful analysis is required to determine the most suitable solution to a specific problem so as to ensure accuracy in future studies.

Appendix

A link to the GitHub repository containing our code can be found here.

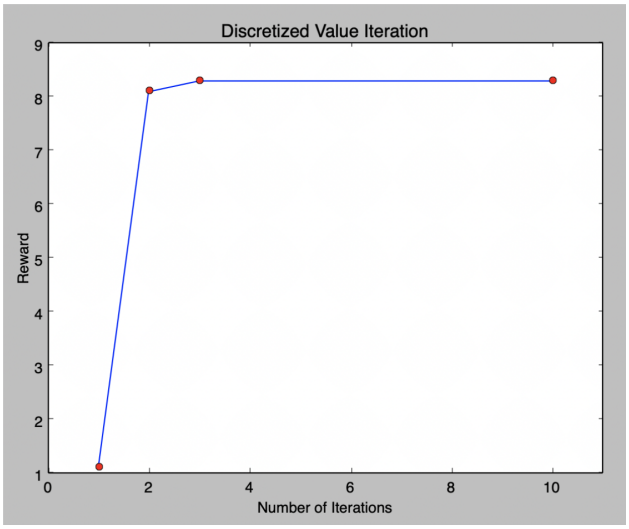

Fig. 4 Discrete Value Iteration Run with Varying Number of Iterations

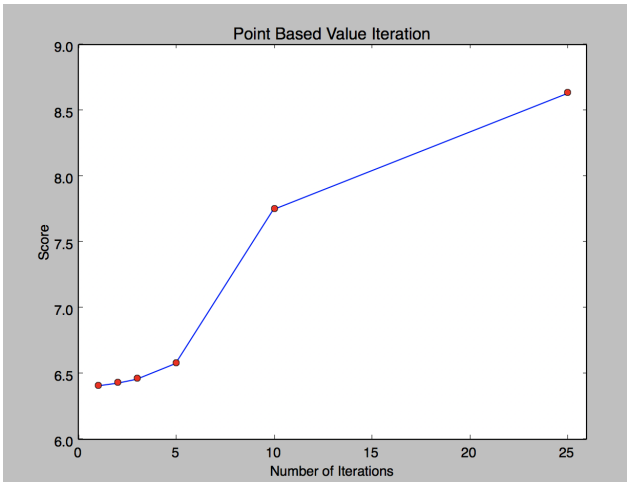

Fig. 5 Randomized Point-Based Value Iteration Run with Varying Number of Iterations

References

[1] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R., "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, Vol. 101, No. 1-2, 1998, p. 99–134. doi:10.1016/s0004-3702(98)00023-x.

[2] Zilberstein, S., "Building Strong Semi-Autonomous Systems," *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence*, Austin, Texas, 2015, pp. 4088–4092. URL http://rbr.cs.umass.edu/shlomo/papers/Zaaai15.html.

[3] Pineau, J., Gordon, G., and Thrun, S., "Point-based value iteration: An anytime algorithm for POMDPs," *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003, pp. 1025 – 1032.

[4] Spaan, M. T., and Vlassis, N., "Perseus: Randomized Point-based Value Iteration for POMDPs," *Journal of Artificial Intelligence Research*, Vol. 24, 2005, p. 195–220. doi:10.1613/jair.1659.

[5] Shani, G., Brafman, R. I., and Shimony, S. E., "Forward Search Value Iteration for POMDPs," *IJCAI*, 2007.

[6] Pinheiro, P., Cardozo, E., Wainer, J., and Rohmer, E., "Cleaning Task Planning for an Autonomous Robot in Indoor Places with Multiples Rooms," *International Journal of Machine Learning and Computing*, Vol. 5, No. 2, 2015, p. 86–90. doi: 10.7763/ijmlc.2015.v5.488.

[7] Kochenderfer, M. J., *Decision making under uncertainty: theory and application*, The MIT Press, 2015.

[8] Sunberg, Z., and Kochenderfer, M. J., "POMCPOW: An online algorithm for POMDPs with continuous state, action, and observation spaces," *CoRR*, Vol. abs/1709.06196, 2017. URL http://arxiv.org/abs/1709.06196.

Group Contribution

Group Contribution: We each implemented a subset of the algorithms included in the research paper. Ana performed the exact solution Value Iteration Method and the Monte Carlo Tree search after experimentation with the QMDP and Fast Informed Bound solutions (which we decided to abandon). Allison built the offline Point-Based Value Iteration using a combination of the provided source code and her own Julia implementation. Ali worked on the online implementation of POMCPOW after experimentation with forward search (from scratch) and ARDESPOT. All three of us worked on the paper, writing up our analysis of our individual algorithm and then dividing the remaining sections of the paper equally amongst ourselves.